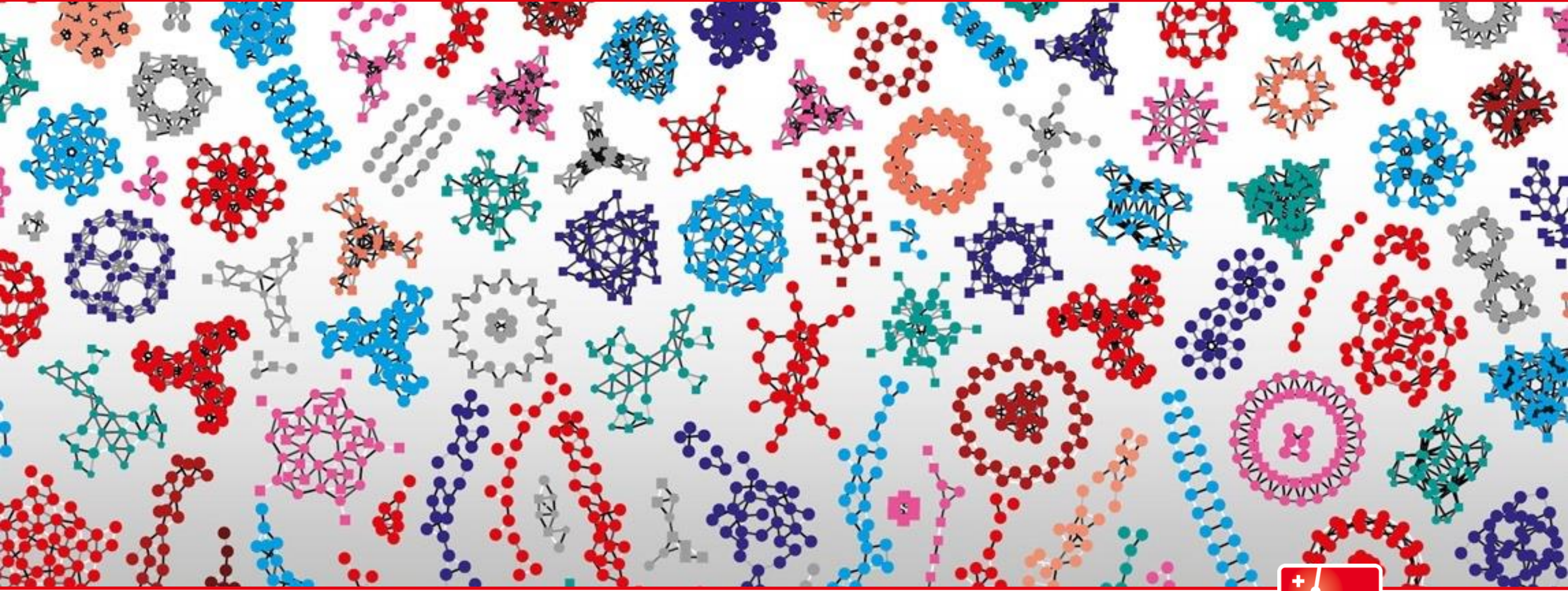


Swiss Institute of Bioinformatics

# Advanced R
# September 2024

Frédéric Schütz (Frederic.Schutz@sib.swiss)
Frédéric Burdet (Frederic.Burdet@sib.swiss)

## An introduction (or reminder) about R data structures

## *Types*

All objects in R have a **type**, which describes the type of data stored in the object.

## *The* `typeof()` *command*

To find the type of any object:

```
typeof(object)
```

```
> typeof( c(1,2,3) )
[1] "double"
> typeof( c("a", "b", "c") )
[1] "character"
```

## *Possible types in R*

- logical
- integer
- double
- closure
- builtin
- special
- complex
- character
- raw
- list

(and a few others)

Sometimes, we also talk about the **mode**, a simplified version of types.

## *The* `mode()` *command*

To find the mode of any object:

```
mode(object)
```

```
> typeof( c(1,2,3) )
[1] "double"
> typeof( c("a", "b", "c") )
[1] "character"

> mode( c(1,2,3) )
[1] "numeric"
> mode( c("a", "b", "c") )
[1] "character"
```

## *Possible types in R*

**Type**

- logical
- integer
- double
- closure
- builtin
- special
- complex
- character
- raw
- list

## *Possible types and modes in R*

| **Type** | **Mode** |
|---|---|
| • logical | logical |
| • integer | numeric |
| • double | numeric |
| • closure | function |
| • builtin | function |
| • special | function |
| • complex | complex |
| • character | character |
| • raw | raw |
| • list | list |

## *An explanation for some abstruse error messages in R*

```
> x <- 1
> print(c(class(x), mode(x), typeof(x)))
[1] "numeric" "numeric" "double"
> x <- letters
> print(c(class(x), mode(x), typeof(x)))
[1] "character" "character" "character"
> x <- TRUE
> print(c(class(x), mode(x), typeof(x)))
[1] "logical" "logical" "logical"
> print(c(class(x), mode(x), typeof(x)))
[1] "logical" "logical" "logical"
> x <- cars[1]
> print(c(class(x), mode(x), typeof(x)))
[1] "data.frame" "list"       "list"
> x <- cars[[1]]
> print(c(class(x), mode(x), typeof(x)))
[1] "numeric" "numeric" "double"
> x <- matrix(cars)
> print(c(class(x), mode(x), typeof(x)))
[1] "matrix" "array"  "list"   "list"
> x <- new.env()
> print(c(class(x), mode(x), typeof(x)))
[1] "environment" "environment" "environment"
> x <- ls
> print(c(class(x), mode(x), typeof(x)))
[1] "function" "function" "closure"
```

https://stackoverflow.com/questions/35445112/what-is-the-difference-between-mode-and-class-in-r

## *An explanation for some abstruse error messages in R*

| | |
|---|---|
| • logical | logical |
| • integer | numeric |
| • double | numeric |
| • closure | function |

```
> f <- function() {}
> f$a
Error in f$a : object of type 'closure' is not subsettable
```

## Vectors

## *Storing data into R*

The simplest way to store data into R is the vector, which contains an ordered collection of objects **of the same type**:

```
> x <- c(1, 2, 3, 4); x
[1] 1 2 3 4
> typeof(x); mode(x)
[1] "double"
[1] "numeric"
```

## *Storing data into R*

The simplest way to store data into R is the vector, which contains an ordered collection of objects **of the same type**:

```
> x <- c(1, 2, 3, 4); x
[1] 1 2 3 4
> typeof(x); mode(x)
[1] "double"
[1] "numeric"

> x <- c(TRUE, FALSE, TRUE, TRUE)
> typeof(x); mode(x)
[1] "logical"
[1] "logical"
```

## *What happens if I store several types of objects in a vector ?*

```
> x <- c(1, 2, TRUE, 3); x
```

## *What happens if I store several types of objects in a vector ?*

```
> x <- c(1, 2, TRUE, 3); x
[1] 1 2 1 3
> typeof(x)
[1] "double"
```

R will convert the objects to the type that is able to accommodate all of them.

## *What happens if I store several types of objects in a vector ?*

```
> x <- c(1, 2, TRUE, 3); x
[1] 1 2 1 3
> typeof(x)
[1] "double"

> x <- c(1, 2, "true", 4); x
```

## *What happens if I store several types of objects in a vector ?*

```
> x <- c(1, 2, TRUE, 3); x
[1] 1 2 1 3
> typeof(x)
[1] "double"

> x <- c(1, 2, "true", 4); x
[1] "1"    "2"    "true" "4"
> typeof(x)
[1] "character"
```

## *With some surprises…*

```
> x <- c("a", TRUE, 3); x
[1] "a"    "TRUE" "3"
> typeof(x)
[1] "character"
```

## *With some surprises…*

```
> x <- c("a", TRUE, 3); x
[1] "a"    "TRUE" "3"
> typeof(x)
[1] "character"

> x <- c("a", c(TRUE, 3)); x
[1] "a" "1" "3"
> typeof(x)
[1] "character"
```

## *Logical vs numeric*

Logical values (TRUE/FALSE) can easily be converted to numeric value (0/1) and back, as in most programming languages:

```
> as.numeric( c(FALSE, TRUE) )
[1] 0 1
> as.logical( c(0,1) )
[1] FALSE  TRUE

> c(FALSE, 0, TRUE)
[1] 0 0 1
```

## *Counting values using logicals*

This is very useful, for example for counting purposes.

Example: count **the number of elements** of the vector `data` that are larger than zero:

```
> data <- rnorm(10)
> data
 [1] -0.61518461 -0.62574053  1.21586046 -1.42627945
 [5]  0.06749257  0.59811401  0.25876230 -0.45936110
 [9] -1.83171441  0.28693148
> data > 0
 [1] FALSE FALSE  TRUE FALSE  TRUE
 [6]  TRUE  TRUE FALSE FALSE  TRUE
> sum(data > 0)
[1] 5
```

## *Logical vs numeric*

Example: count **the proportion of elements** of the vector `data` that are larger than zero:

```
> data <- rnorm(10)
> data
 [1] -0.61518461 -0.62574053  1.21586046 -1.42627945
 [5]  0.06749257  0.59811401  0.25876230 -0.45936110
 [9] -1.83171441  0.28693148
> data > 0
 [1] FALSE FALSE  TRUE FALSE  TRUE
 [6]  TRUE  TRUE FALSE FALSE  TRUE
> mean(data > 0)
[1] 0.5
```

## *Why do the two selection commands return different results ?*

```
> vector <- 1:10

> vector[ c(0,1) ]
[1] 1
> vector[ c(FALSE, TRUE) ]
[1]  2  4  6  8 10
```

## *Difference between logical and numeric*

```
> vector <- 1:10

> vector[ c(0,1) ]
[1] 1
```

This selects elements 0 (which does not exist) and 1 (=1)

```
> vector[ c(FALSE,TRUE) ]
[1]  2  4  6  8 10
```

This applies to each element in turn; since the logical vector is not long enough, it is recycled to cover the full vector. At the end, only elements at even positions are selected.

## *Difference between logical and numeric*

```
> vector <- 1:10

> vector[ c(0,1) ]
[1] 1
> vector[ c(FALSE, TRUE) ]
[1]  2  4  6  8 10
```

In contrast to other programming languages, logical and numeric types can not be freely exchanged !

## *How to use this  feature to your advantage*

This behaviour can lead to bugs in your code. But if done correctly, it can also help you.

## *How to use this  feature to your advantage*

This behaviour can lead to bugs in your code. But if done correctly, it can also help you.

The following command is a simple way to select all elements at even positions in the vector.

(for example, sampling one data point out of two)

```
> vector[ c(FALSE,TRUE) ]
[1]  2  4  6  8 10
```

## *What could possibly go wrong ?*

```
> sample(1:10, 10, replace=T)
[1]  8  4  9  1 10  3  9  6  3  9
```

## *When* `T` *does not mean* `TRUE`

```
> A <- "a"; B <- "b"; C <- "c"; T <- "t"

> sample(1:10, 10, replace=T)
> A <- "a"; B <- "b"; C <- "c"; T <- "t"

> sample(1:10, 10, replace=T)
Error in sample(1:10, 10, replace = T) : invalid 'replace' argument
```

```
> A <- "a"; B <- "b"; C <- "c"; T <- "t"

> sample(1:10, 10, replace=T)
Error in sample(1:10, 10, replace = T) : invalid 'replace' argument
```

'`T`' and '`F`' can be freely redefined by the user, something impossible with the full form:

```
> TRUE <- "t"
Error in TRUE <- "t" : invalid (do_set) left-hand side to assignment
```

This will yield an error, or even worse…

## *If you are really vicious…*

```
> T <- FALSE

> sample(1:10, 10, replace=T)
 [1]  7  6  3  4 10  1  8  5  9  2
```

## *If you are really vicious…*

```
> T <- FALSE

> sample(1:10, 10, replace=T)
 [1]  7  6  3  4 10  1  8  5  9  2
```

Or, more likely:

```
> T <- complicated_function(many, many, complicated, arguments, and
                            the, function, returns, FALSE, in, the,
                            end )
> sample(1:10, 10, replace=T)
 [1]  7  6  3  4 10  1  8  5  9  2
```

## *Attributes*

**Attributes** are arbitrary labels attached to the R objects.

## *Attributes*

**Attributes** are arbitrary labels attached to the R objects.

```
> x <- rnorm(10)
> attributes(x)
NULL
```

## *Attributes*

**Attributes** are arbitrary labels attached to the R objects.

```
> x <- rnorm(10)
> attributes(x)
NULL
> attr(x, "mylabel") <- "Random normal data"
> attr(x, "mylabel")
[1] "Random normal data"
```

## *Attributes*

**Attributes** are arbitrary labels attached to the R objects.

```
> x <- rnorm(10)
> attributes(x)
NULL
> attr(x, "mylabel") <- "Random normal data"
> attr(x, "mylabel")
[1] "Random normal data"
> attributes(x)
$mylabel
[1] "Random normal data"
```

## *Some important attributes in R*

- **names**: allows naming the components of an object
- **class:** a label attached to the object, which indicates how actions can be performed on the object
- **dim:** the dimensions of the objects (e.g. for a matrix or an array)

## *Adding a class*

```
> x <- rnorm(10)
> attributes(x)
NULL
> attr(x, "mylabel") <- "Random normal data"
> attr(x, "mylabel")
[1] "Random normal data"
> attributes(x)
$mylabel
[1] "Random normal data"

> class(x) <- "randomdata"
```

## *Adding a class*

```
> x <- rnorm(10)
> attributes(x)
NULL
> attr(x, "mylabel") <- "Random normal data"
> attr(x, "mylabel")
[1] "Random normal data"
> attributes(x)
$mylabel
[1] "Random normal data"

> class(x) <- "randomdata"
> attr(x, "class") <- "randomdata"   # equivalent
```

## *Adding a class*

```
> x <- rnorm(10)
> attributes(x)
NULL
> attr(x, "mylabel") <- "Random normal data"
> attr(x, "mylabel")
[1] "Random normal data"
> attributes(x)
$mylabel
[1] "Random normal data"

> class(x) <- "randomdata"
> attr(x, "class") <- "randomdata"    # equivalent
> class(x)
[1] "randomdata"
```

## *Adding names*

```
> names(x) <- LETTERS[1:10]
> x
          A           B           C           D           E 
-0.93205027 -0.16194958  0.26727310 -0.07427123  1.54048877 
          F           G           H           I           J 
-0.63579513  0.27141749 -2.03039854 -2.52658864  1.02263626 
attr(,"mylabel")
[1] "Random normal data"
attr(,"class")
[1] "randomdata"

> attributes(x)
$mylabel
[1] "Random normal data"

$class
[1] "randomdata"

$names
[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

## *Adding names*

```
> names(x) <- LETTERS[1:10]
> x
         A          B          C          D          E 
-0.93205027 -0.16194958  0.26727310 -0.07427123  1.54048877 
         F          G          H          I          J 
-0.63579513  0.27141749 -2.03039854 -2.52658864  1.02263626 
attr(,"mylabel")
[1] "Random normal data"
attr(,"class")
[1] "randomdata"

> attributes(x)
$mylabel
[1] "Random normal data"

$class
[1] "randomdata"

$names
[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

## Names allow you to create **lookup tables**

```
> names(x) <- LETTERS[1:10]
> x
          A           B           C           D           E 
-0.93205027 -0.16194958  0.26727310 -0.07427123  1.54048877 
          F           G           H           I           J 
-0.63579513  0.27141749 -2.03039854 -2.52658864  1.02263626 

> x["B"]
[1] -0.16194958
```

Reproducible Research

## Names allow you to create **lookup tables**

```
> names(x) <- LETTERS[1:10]
> x
          A           B           C           D           E 
-0.93205027 -0.16194958  0.26727310 -0.07427123  1.54048877 
          F           G           H           I           J 
-0.63579513  0.27141749 -2.03039854 -2.52658864  1.02263626 

> x["B"]
[1] -0.16194958
> x[2]
[1] -0.16194958
```

## Matrices and arrays

Matrices (in 2D) and arrays (in 2D or more) are extensions of vectors, where two or more dimensions are specified.

```
> m <- matrix(1:30, ncol=6)
> m
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    6   11   16   21   26
[2,]    2    7   12   17   22   27
[3,]    3    8   13   18   23   28
[4,]    4    9   14   19   24   29
[5,]    5   10   15   20   25   30

> m[1,3]
[1] 11
```

## *Arrays and matrices*

In fact, a matrix (or array) is stored as a vector (column by column) with additional information about its dimensions.

```
> m <- matrix(1:30, ncol=6)
> m
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    6   11   16   21   26
[2,]    2    7   12   17   22   27
[3,]    3    8   13   18   23   28
[4,]    4    9   14   19   24   29
[5,]    5   10   15   20   25   30
> as.vector(m)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
[16] 17 18 19 20 21 22 23 24 25 26 26 27 28 29 30
```

## *Arrays and matrices*

Internally, matrices are just vectors, with indications of dimensions.

```
> m <- matrix(1:30, ncol=6)
> m
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    6   11   16   21   26
[2,]    2    7   12   17   22   27
[3,]    3    8   13   18   23   28
[4,]    4    9   14   19   24   29
[5,]    5   10   15   20   25   30
> as.vector(m)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
[16] 17 18 19 20 21 22 23 24 25 26 26 27 28 29 30
```

## *Matrices and vectors*

So you can also access matrices as if they were vectors.

```
> m <- matrix(1:30, ncol=6)

> m[11]; m[1,3]                    # Equivalent
[1] 11
[1] 11
```

## *Matrices and vectors*

They have both a **length** and **dimensions**.

```
> m <- matrix(1:30, ncol=6)

> m[11]; m[1,3]                    # Equivalent
[1] 11
[1] 11
> dim(m)
[1] 5 6
> length(m)
[1] 30
```

Arrays are constructed in a similar way.

```
> a <- 1:24
> array(a, dim=c(4,3,2))
, , 1

     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12

, , 2

     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24
```

## *Transforming a vector into a matrix*

You must specify the dimensions and change the class.

```
> a <- 1:30
> attr(a, "dim") <- c(5,6)
> class(a) <- "matrix"
> a
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    6   11   16   21   26
[2,]    2    7   12   17   22   27
[3,]    3    8   13   18   23   28
[4,]    4    9   14   19   24   29
[5,]    5   10   15   20   25   30
```

## *Creating a matrix row by row*

A matrix can also be created row by row, using the `byrow` parameter.

However, it will still be stored column by column.

```
> m <- matrix(1:30, ncol=6, byrow=TRUE); m
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    2    3    4    5    6
[2,]    7    8    9   10   11   12
[3,]   13   14   15   16   17   18
[4,]   19   20   21   22   23   24
[5,]   25   26   27   28   29   30


> as.vector(m)
  [1]  1  7 13 19 25  2  8 14 20 26  3  9 15 21 27
 [16]  4 10 16 22 28  5 11 17 23 29  6 12 18 24 30
```

## *Type of elements in a matrix*

Since they are vectors, all elements of matrices must be of the same type:

```
> m <- matrix(1:30, ncol=6)
> typeof(m)
[1] "integer"
> m[3,3] <- "a"
> m
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,] "1"  "6"  "11" "16" "21" "26"
[2,] "2"  "7"  "12" "17" "22" "27"
[3,] "3"  "8"  "a"  "18" "23" "28"
[4,] "4"  "9"  "14" "19" "24" "29"
[5,] "5"  "10" "15" "20" "25" "30"
> typeof(m)
[1] "character"
```

```
selectcolumns <- function( m, cols, rows ) {
  m1 <- m [, cols]
  m2 <- m1[rows, ]
  m2
}

nrows <- 20
m1 <- data.frame( a=runif(nrows), b=runif(nrows), c=runif(nrows) )
row.names(m1) <- paste0("row", 1:nrow(m1))

cols <- c("b", "c")
rows <- c("row10", "row12")

> selectcolumns(m1, cols, rows)
              b         c
row10 0.8578518 0.2864960
row12 0.3767570 0.7874534
```

```
selectcolumns <- function( m, cols, rows ) {
  m1 <- m [, cols]
  m2 <- m1[rows, ]
  m2
}

nrows <- 20
m1 <- data.frame( a=runif(nrows), b=runif(nrows), c=runif(nrows) )
row.names(m1) <- paste0("row", 1:nrow(m1))

cols <- "b"
rows <- c("row10", "row12")

> selectcolumns(m1, cols, rows)
```

```
selectcolumns <- function( m, cols, rows ) {
  m1 <- m [, cols]
  m2 <- m1[rows, ]
  m2
}

nrows <- 20
m1 <- data.frame( a=runif(nrows), b=runif(nrows), c=runif(nrows) )
row.names(m1) <- paste0("row", 1:nrow(m1))

cols <- "b"
rows <- c("row10", "row12")

> selectcolumns(m1, cols, rows)
Error in m1[rows, ] : incorrect number of dimensions
```

## *Subsetting a matrix*

```
> m <- matrix(1:6, nrow=2)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

## *Subsetting a matrix*

```
> m <- matrix(1:6, nrow=2)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> m[,1:2]
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

… yields a matrix.

## *Subsetting a matrix*

```
> m <- matrix(1:6, nrow=2)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> m[,1]
```

## *Subsetting a matrix*

```
> m <- matrix(1:6, nrow=2)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

> m[,1]
[1] 1 2
```

… yields a **vector** (instead of a 2 x 1 matrix).

By default, R removes all dimensions
that it deems not useful !

To avoid this, use the `drop=FALSE` option to the matrix subsetting:

```
> m[,1]
[1] 1 2

> m[, 1, drop=FALSE]
     [,1]
[1,]    1
[2,]    2

> m[1, , drop=FALSE]
     [,1] [,2] [,3]
[1,]    1    3    5
```

It is not possible to set `drop=FALSE` as the default mode.

Doing this would mean that accessing one element in a matrix would return a 1x1 matrix:

```
> m[2,3, drop=FALSE]
        [,1]
   [1,]    6
```

… which is almost certainly not what you want.

## *Another possible consequence*

```
> head(data1, 3)
  identifier    var1    var2
1       3862 0.87207 -2.0105
2       1577 0.01075  0.1970
3       5150 1.28249 -0.4650
> head(data2, 3)
  identifier   var3    var4
1       3862 0.1383 -2.0165
2       1577 2.3219  0.6855
3       5150 0.6865  0.7783
> data <- cbind( data1[, c("var1", "var2")],
                 data2[, c("var3", "var4")],
                 data1[, "identifier"] )
```

## *Matrices converted to vectors lose their names !*

```
> head(data1, 3)
  identifier    var1    var2
1       3862 0.87207 -2.0105
2       1577 0.01075  0.1970
3       5150 1.28249 -0.4650
> head(data2, 3)
  identifier   var3    var4
1       3862 0.1383 -2.0165
2       1577 2.3219  0.6855
3       5150 0.6865  0.7783
> data <- cbind( data1[, c("var1", "var2")],
                 data2[, c("var3", "var4")],
                 data1[, "identifier"] )
> head(data, 3)
     var1     var2     var3    var4 data1[, "identifier"]
1 0.87207 -2.01057  0.13836 -2.0165                  3862
2 0.01075  0.19709  2.32192  0.6855                  1577
3 1.28249 -0.46507  0.68659  0.7783                  5150
```

## *Matrices converted to vectors lose their names !*

```
> head(data1, 3)
  identifier    var1    var2
1       3862 0.87207 -2.0105
2       1577 0.01075  0.1970
3       5150 1.28249 -0.4650
> head(data2, 3)
  identifier   var3    var4
1       3862 0.1383 -2.0165
2       1577 2.3219  0.6855
3       5150 0.6865  0.7783
> data <- cbind( data1[, c("var1", "var2")],
               data2[, c("var3", "var4")],
               data1[, "identifier", drop=FALSE] )
> head(data, 3)
      var1     var2     var3    var4 identifier
1  0.87207 -2.01057  0.13836 -2.0165       3862
2  0.01075  0.19709  2.32192  0.6855       1577
3  1.28249 -0.46507  0.68659  0.7783       5150
```

## *What happens if I store several types of objects in a vector ?*

```
> x <- c(1, "a", c); x
```

## *What happens if I store several types of objects in a vector ?*

```
> x <- c(1, "a", c); x
```

## *What happens if I store several types of objects in a vector ?*

```
> x <- c(1, "a", c); x
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
function (...)  .Primitive("c")

> typeof(x)
[1] "list"
```

## *Lists*

Lists allow the storage of several objects (with different types) in a single R object.

```
> mylist <- list(ages=c(21, 32, 41, 45),
                 height=c(180, 176, 156, 165),
                 sex=c("M", "M", "F", "M") )
```

## *Lists*

Lists allow the storage of several objects (with different types) in a single R object.

```
> mylist <- list(ages=c(21, 32, 41, 45),
                 height=c(180, 176, 156, 165),
                 sex=c("M", "M", "F", "M") )
> mylist
$ages
[1] 21 32 41 45

$height
[1] 180 176 156 165

$sex
[1] "M" "M" "F" "M"
```

## *Lists*

Lists allow the storage of several objects (with different types) in a single R object.

```
> mylist <- list(ages=c(21, 32, 41, 45),
                 height=c(180, 176, 156, 165),
                 sex=c("M", "M", "F", "M") )
> mylist
$ages
[1] 21 32 41 45

$height
[1] 180 176 156 165

$sex
[1] "M" "M" "F" "M`

> class(mylist); typeof(mylist)
[1] "list"
[1] "list"
```

**When accessing a list element, what is the difference between**

**`mylist[1]`**

**and**

**`mylist[[1]] ?`**

The objects can be accessed either using their rank, or by their name.

| | |
|---|---|
| `[x]` | returns part (one element) of the list |
| `[[x]]` | returns what is inside this element |

The objects can be accessed either using their rank, or by their name.

| | |
|---|---|
| `[x]` | returns part (one element) of the list |
| `[[x]]` | returns what is inside this element |

```
> mylist[1]
```

## *Lists*

The objects can be accessed either using their rank, or by their name.

| | |
|---|---|
| `[x]` | returns part (one element) of the list |
| `[[x]]` | returns what is inside this element |

```
> mylist[1]
$ages
[1] 21 32 41 45
> typeof(mylist[1])
[1] "list"
```

The objects can be accessed either using their rank, or by their name.

| | |
|---|---|
| `[x]` | returns part (one element) of the list |
| `[[x]]` | returns what is inside this element |

```
> mylist[1]
$ages
[1] 21 32 41 45
> typeof(mylist[1])
[1] "list"

> mylist[[1]]
[1] 21 32 41 45
> typeof(mylist[[1]])
[1] "double"
```

The objects can be accessed either using their rank, or by their name.

| | |
|---|---|
| `[x]` | returns part (one element) of the list |
| `[[x]]` | returns what is inside this element |

```
> mylist[1]
$ages
[1] 21 32 41 45
> typeof(mylist[1])
[1] "list"

> mylist[[1]]
[1] 21 32 41 45
> typeof(mylist[[1]])
[1] "double"

> mylist$height
[1] 180 176 156 165
```

## *Vectors vs lists*

- **Atomic vectors**:
  an ordered collection of data
  of the **same type**

- **Lists:**
  an ordered collection of data
  that can be of **different types**.

## Data frames

## *Data frames*

Data frames are usually the preferred method for working with datasets that consists of several observations (rows) on several variables (columns).

```
> data <- as.data.frame( mylist )
> data
  ages height sex
1   21    180   M
2   32    176   M
3   41    156   F
4   45    165   M
```

4 observations

3 variables

## *Data frames*

```
> data <- as.data.frame( mylist )
> data
  ages height sex
1   21    180   M
2   32    176   M
3   41    156   F
4   45    165   M
```

## *Data frames*

```
> data <- as.data.frame( mylist )
> data
  ages height sex
1   21    180   M
2   32    176   M
3   41    156   F
4   45    165   M
> class(data); typeof(data)
```

## *Data frames are lists that look like matrices*

Data frames are actually lists in R.

```
> data <- as.data.frame( mylist )
> data
  ages height sex
1   21    180   M
2   32    176   M
3   41    156   F
4   45    165   M
> class(data); typeof(data)
[1] "data.frame"
[1] "list"
```

## *Data frames are lists that look like matrices*

They are easier to use than lists: you can access the elements as in a matrix all elements, since they all have the same length

```
> data <- as.data.frame( mylist )
> data
  ages height sex
1   21    180   M
2   32    176   M
3   41    156   F
4   45    165   M
> data[2,2]
[1] 176
```

## *Data frames are lists that look like matrices*

They are easier to use than lists: you can access the elements as in a matrix all elements, since they all have the same length

They are more flexible than matrices, as they allow columns of differents types, while still making them easy to access.

```
> data <- as.data.frame( mylist )
> data
  ages height sex
1   21    180   M
2   32    176   M
3   41    156   F
4   45    165   M
> typeof(data[,1]); typeof(data[,3])
[1] "double"
[1] "character"
```

## *Data frames vs lists*

To convert a list into a matrix, one only needs to:

- change the class to `data.frame`
- give (unique) names to the rows by setting the `row.names` attribute

```
> class(mylist) <- "data.frame"
```

## *Data frames vs lists*

To convert a list into a matrix, one only needs to:

- change the class to `data.frame`
- give (unique) names to the rows by setting the `row.names` attribute

```
> class(mylist) <- "data.frame"
> mylist
[1] ages   height sex
<0 rows> (or 0-length row.names)
```

## *Data frames vs lists*

To convert a list into a matrix, one only needs to:

- change the class to `data.frame`
- give (unique) names to the rows by setting the `row.names` attribute

```
> class(mylist) <- "data.frame"
> mylist
[1] ages   height sex
<0 rows> (or 0-length row.names)
> row.names(mylist) <- 1:length(mylist[[1]])
> mylist
  ages height sex
1   21     180   M
2   32     176   M
3   41     156   F
4   45     165   M
```

## *Data frames: accessing columns*

```
> data
  ages height sex
1   21    180   M
2   32    176   M
3   41    156   F
4   45    165   M

> data[1]
  ages
1   21
2   32
3   41
4   45
```

Columns can be accessed just like a list

The result is a **single-column dataframe**.

## *Data frames: accessing columns*

```
> data
  ages height sex
1   21    180   M
2   32    176   M
3   41    156   F
4   45    165   M

> data[,1]
[1] 21 32 41 45

> data[[1]]
[1] 21 32 41 45
```

Alternatively, it is possible to access the **content** of a given column, yielding a vector.

## *Data frames: accessing columns by names*

Columns can also be accessed by name:

```
> data$height
[1] 180 176 156 165

> data[, "height"]
[1] 180 176 156 165
```

This is usually better than accessing them by column number, as the name is less likely to change than the column number.

(also, if the name changes, it will yield an error)

## *Data frames: accessing columns by names*

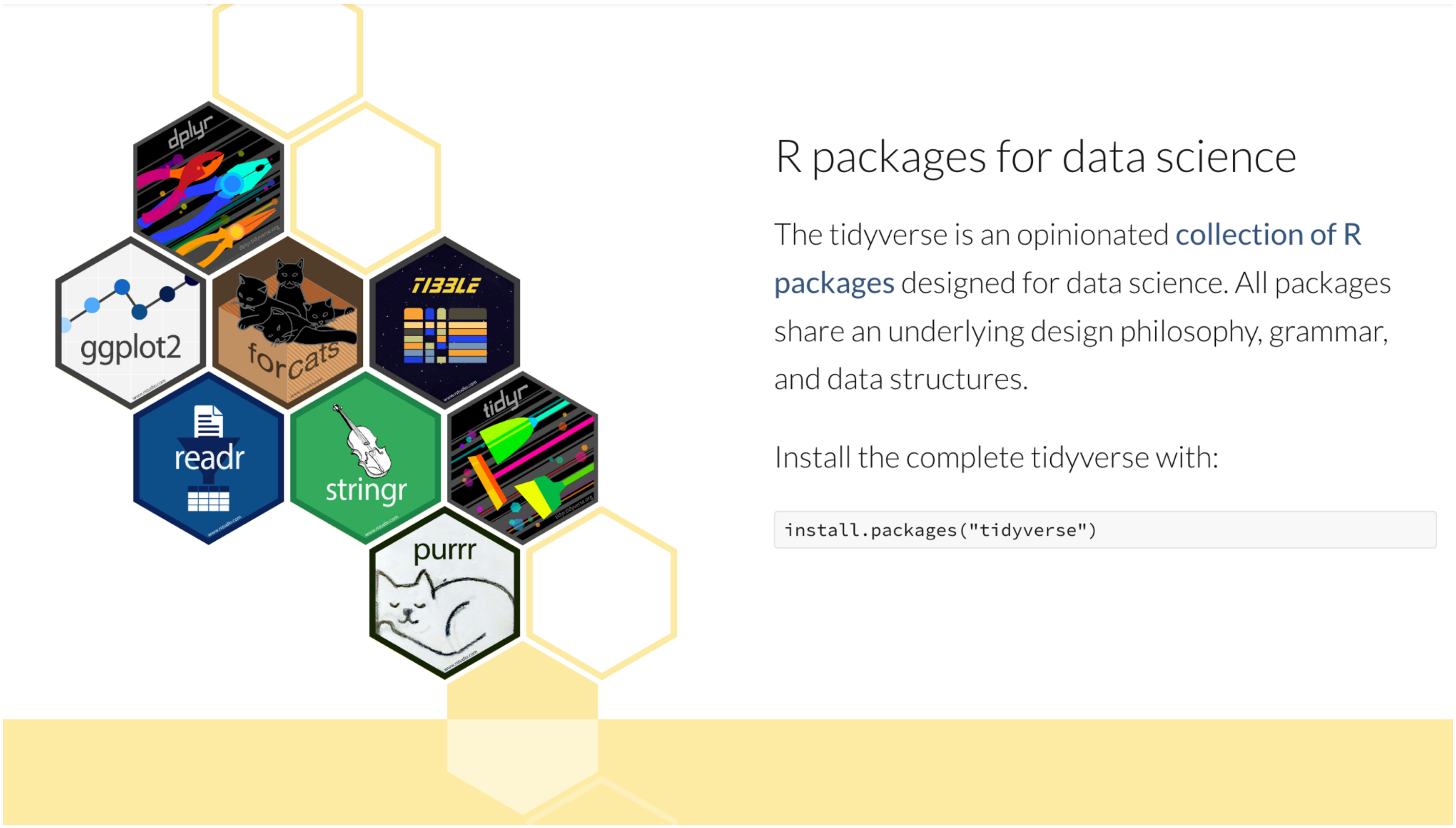
You can shorten the name as long as there is no ambiguity:

```
> data$height
[1] 180 176 156 165

> data$h
[1] 180 176 156 165
```

This is not recommended: the code may break if your script is used on a dataset that includes a new column which causes an ambiguity.

## *tidyverse is a collection of packages*



R packages for data science

The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

Install the complete tidyverse with:

```
install.packages("tidyverse")
```

## *dplyr is a grammar of data manipulation*

- Takes a data frame (or tibble) as the first argument
- Several features available:
  - mutate() adds new variables that are functions of existing variables
  - select() picks variables based on their names.
  - filter() picks cases based on their values.
  - summarise() / reframe() reduce multiple values down to a single summary.
  - arrange() changes the ordering of the rows.
- Features like connecting directly to a database, or work on data not fully loaded in memory
- Functions like bind_rows and bind_cols much more efficient than rbind and cbind !

https://dplyr.tidyverse.org/articles/dplyr.html

*Naive data manipulation in base R*

Take the iris data.frame, derive a new variable and calculate an average by group

```
> data(iris)
> head(iris, n = 2)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
> iris$Length.Product <- iris$Sepal.Length * iris$Petal.Length
> iris <- iris[iris$Length.Product >= 7, ]
> iris <- iris[, c("Species", "Length.Product")]
> sqldf("select Species, avg(`Length.Product`) as
avg_length_product from iris group by Species")
     Species avg_length_product
1     setosa           7.922727
2 versicolor          25.466600
3  virginica          36.873800
```

*Now with dplyr*

Take the iris data.frame, derive a new variable and calculate an average by group

```
iris.modified <-
  iris |>
  mutate(Length.Product = Sepal.Length * Petal.Length) |>
  filter(Length.Product >= 7) |>
  select(Species, Length.Product) |> # optional
  reframe(avg.length.product = mean(Length.Product), .by =
Species)

 Species avg.length.product
1     setosa           7.922727
2 versicolor          25.466600
3  virginica          36.873800
```

## Some useful features like everything() (see also relocate() )

```
> data(iris)
> iris <- iris |> mutate(Length.Product = Sepal.Length * 
Petal.Length)
> head(iris[, 1:4], n = 2)
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1          5.1         3.5          1.4         0.2
2          4.9         3.0          1.4         0.2
> iris <- iris |> select(Species, Length.Product, everything())
> head(iris[, 1:4], n = 2)
  Species Length.Product Sepal.Length Sepal.Width
1  setosa           7.14          5.1         3.5
2  setosa           6.86          4.9         3.0
```

## Or range of columns using the names

```
> head(iris |> select(Sepal.Length:Petal.Width))
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1          5.1         3.5          1.4         0.2
2          4.9         3.0          1.4         0.2
3          4.7         3.2          1.3         0.2
4          4.6         3.1          1.5         0.2
5          5.0         3.6          1.4         0.2
6          5.4         3.9          1.7         0.4
```

## *Getting information about R objects:* `summary()`

The **`summary()`** command gives some brief information about an R object.

```
> summary(mylist)
       Length Class  Mode
ages   4      -none- numeric
height 4      -none- numeric
sex    4      -none- character
```

## *Getting information about R objects:* `summary()`

Its output depends on the type of object:

```
> summary(mylist)
       Length Class  Mode
ages   4      -none- numeric
height 4      -none- numeric
sex    4      -none- character

> summary( rnorm(100) )
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-2.62861 -0.49714  0.14360  0.05439  0.69587  2.11127
```

*Getting information about R objects*

The `str()` command gives detailed information about the **str**ucture of an R object:

```
> str(mylist)
List of 3
 $ ages  : num [1:4] 21 32 41 45
 $ height: num [1:4] 180 176 156 165
 $ sex   : chr [1:4] "M" "M" "F" "M"
```

## *Getting information about R objects*

The information provided by `str()` can indeed be really detailed; try the following commands :

```
> model <- lm( runif(10) ~ rnorm(10) )
> str(model)
```

## *Getting information about R objects*

```
# Try this one if you don’t believe the word "detailed" above
> model <- lm( runif(10) ~ rnorm(10) )
> str(model)
List of 12
 $ coefficients : Named num [1:2] 0.5486 0.0335
  ..- attr(*, "names")= chr [1:2] "(Intercept)" "rnorm(10)"
 $ residuals    : Named num [1:10] -0.255582 -0.192832 -0.000517 0.340288 -0.336684 ...
  ..- attr(*, "names")= chr [1:10] "1" "2" "3" "4" ...
 $ effects      : Named num [1:10] -1.744 -0.1102 0.0902 0.4255 -0.2811 ...
  ..- attr(*, "names")= chr [1:10] "(Intercept)" "rnorm(10)" "" "" ...
 $ rank         : int 2
 $ fitted.values: Named num [1:10] 0.521 0.565 0.573 0.568 0.538 ...
  ..- attr(*, "names")= chr [1:10] "1" "2" "3" "4" ...
 $ assign       : int [1:2] 0 1
 $ qr           :List of 5
  ..$ qr   : num [1:10, 1:2] -3.162 0.316 0.316 0.316 0.316 ...
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : chr [1:10] "1" "2" "3" "4" ...
  .. .. ..$ : chr [1:2] "(Intercept)" "rnorm(10)"
  .. ..- attr(*, "assign")= int [1:2] 0 1
  ..$ qraux: num [1:2] 1.32 1.19
  ..$ pivot: int [1:2] 1 2
  ..$ tol  : num 1e-07
  ..$ rank : int 2
  ..- attr(*, "class")= chr "qr"
 $ df.residual  : int 8
 $ xlevels      : Named list()
 $ call         : language lm(formula = runif(10) ~ rnorm(10))
 $ terms        :Classes 'terms', 'formula'  language runif(10) ~ rnorm(10)
  .. ..- attr(*, "variables")= language list(runif(10), rnorm(10))
  .. ..- attr(*, "factors")= int [1:2, 1] 0 1
  .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. ..$ : chr [1:2] "runif(10)" "rnorm(10)"
  .. .. .. ..$ : chr "rnorm(10)"
  .. ..- attr(*, "term.labels")= chr "rnorm(10)"
  .. ..- attr(*, "order")= int 1
  .. ..- attr(*, "intercept")= int 1
  .. ..- attr(*, "response")= int 1
  .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
  .. ..- attr(*, "predvars")= language list(runif(10), rnorm(10))
  .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
  .. .. ..- attr(*, "names")= chr [1:2] "runif(10)" "rnorm(10)"
 $ model        :'data.frame':  10 obs. of  2 variables:
  ..$ runif(10): num [1:10] 0.266 0.372 0.573 0.908 0.202 ...
  ..$ rnorm(10): num [1:10] -0.82 0.487 0.738 0.576 -0.305 ...
  ..- attr(*, "terms")=Classes 'terms', 'formula'  language runif(10) ~ rnorm(10)
  .. .. ..- attr(*, "variables")= language list(runif(10), rnorm(10))
  .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
  .. .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. .. ..$ : chr [1:2] "runif(10)" "rnorm(10)"
  .. .. .. .. ..$ : chr "rnorm(10)"
  .. .. ..- attr(*, "term.labels")= chr "rnorm(10)"
  .. .. ..- attr(*, "order")= int 1
  .. .. ..- attr(*, "intercept")= int 1
  .. .. ..- attr(*, "response")= int 1
  .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
  .. .. ..- attr(*, "predvars")= language list(runif(10), rnorm(10))
  .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
  .. .. .. ..- attr(*, "names")= chr [1:2] "runif(10)" "rnorm(10)"
 - attr(*, "class")= chr "lm"
```

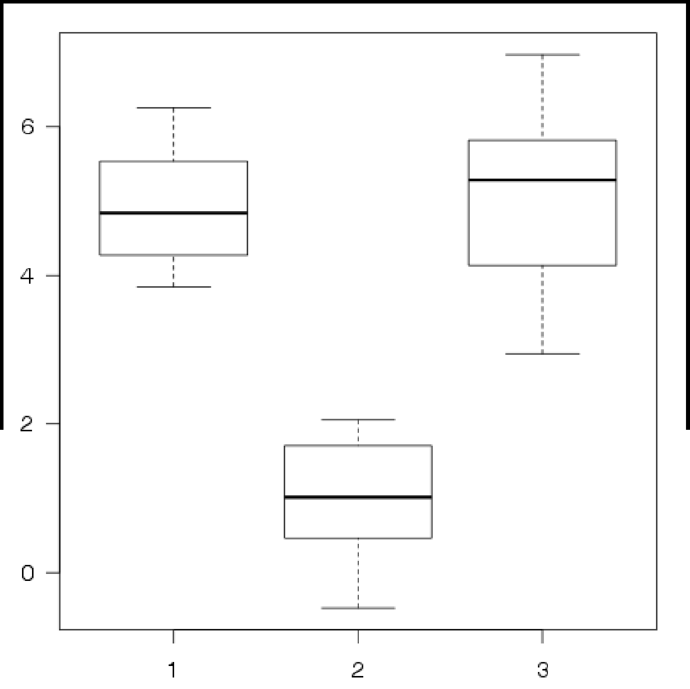
## A question…

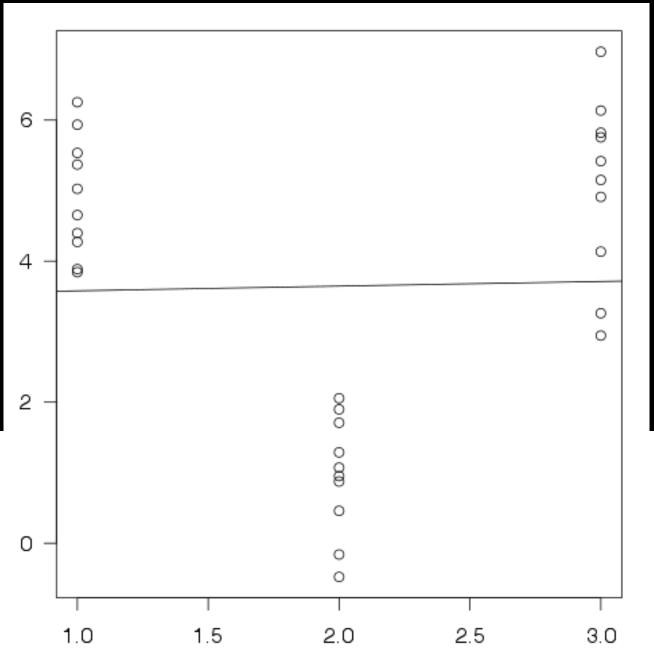
```r
# Simulate data for 3 groups
set.seed(1)
groups <- rep( 1:3, each=10 )

measure <- vector(length=30)
measure[ groups==1 ] <- 5
measure[ groups==2 ] <- 1
measure[ groups==3 ] <- 5
measure <- measure + rnorm(30)

# Perform a one-way ANOVA on this data
boxplot( measure ~ groups )
summary( aov( measure ~ groups ) )
```

```
# Perform a one-way ANOVA on this data
> boxplot( measure ~ groups )
> summary(aov( measure ~ groups ))
            Df Sum Sq Mean Sq F value Pr(>F)
groups       1   0.09   0.088   0.018  0.893
Residuals   28 134.85   4.816
```





## Factors

## *Factors*

Factors represent **categorical variables** in R.

They are vectors that can contain only values from a (finite) predefined set.

## *Example*

```
> hair <- factor(c("blond", "brown", "red", "blond"))
```

## *Example*

```
> hair <- factor(c("blond", "brown", "red", "blond"))

> hair
[1] blond brown red   blond
Levels: blond brown red
```

## *Example*

```
> hair <- factor(c("blond", "brown", "red", "blond"))

> hair
[1] blond brown red   blond
Levels: blond brown red

> hair[2] <- "blond"
> hair
[1] blond blond red   blond
Levels: blond brown red
```

## *Example*

```
> hair <- factor(c("blond", "brown", "red", "blond"))

> hair
[1] blond brown red   blond
Levels: blond brown red

> hair[2] <- "blond"
> hair[2] <- "grey"
Warning message:
In `[<-.factor`(`*tmp*`, 2, value = "grey") :
  invalid factor level, NAs generated
```

## *Example*

```
> hair <- factor(c("blond", "brown", "red", "blond"))

> hair
[1] blond brown red   blond
Levels: blond brown red

> hair[2] <- "blond"
> hair[2] <- "grey"
Warning message:
In `[<-.factor`(`*tmp*`, 2, value = "grey") :
  invalid factor level, NAs generated
> hair
[1] blond <NA>  red   blond
Levels: blond brown red
```

```
> class(hair)
[1] "factor"
> typeof(hair); mode(hair)
[1] "integer"
[1] "numeric"

> as.numeric(hair)
[1]  1 NA  3  1
> as.character(hair)
[1] "blond" NA      "red"   "blond"
```

Internally, R stores factors as integer numbers, along with the correspondance between number and labels (1=blond, 2=brown, 3=red).

## *Ordered factors*

Use the `ordered=TRUE` option for ordinal (ordered) values:

```
> time <- factor(c(1,2,3,2,2,1), levels=c(1,2,3),
                 labels=c("never", "sometimes", "always"),
                 ordered=TRUE)
> time
[1] never     sometimes always    sometimes
[5] sometimes never
Levels: never < sometimes < always
```

## *Some R functions respect ordered factors*

Comparisons work as expected:

```
> time
[1] never     sometimes always    sometimes
[5] sometimes never
Levels: never < sometimes < always
> time[2] < time[3]
[1] TRUE
> "sometimes" < "always"
[1] FALSE
```

Some statistical modelling or plotting functions can adapt their parameters for ordered factors.

```
# Perform a one-way ANOVA on this data
> boxplot( measure ~ groups )
> summary(aov( measure ~ groups ) )
            Df Sum Sq Mean Sq F value Pr(>F)
groups       1   0.09   0.088   0.018  0.893
Residuals   28 134.85   4.816

> groups <- as.factor(groups)
> groups
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
Levels: 1 2 3
> summary(aov( measure ~ groups ) )
            Df Sum Sq Mean Sq F value   Pr(>F)
groups       2  94.12   47.06   52.95 4.53e-10 ***
Residuals   27  24.00    0.89
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Recent versions of R have introduced several major changes with regards to how factors are handled.

## *Concatenating factors: before R 4.1*

```
> c(hair, hair)
[1] 1 2 3 1 1 2 3 1

# Workaround #1
> factor( as.character(hair), as.character(hair) )

# Workaround #2
> unlist( list( hair, hair) )
```

Simply concatenating factors was creating a vector made out of the numeric values, which is almost certainly not what you want.

*Since R 4.1*

This behaviour has changed starting in R version 4.1 (May 2021)

Changelog:

```
Using c() to combine a factor with other
factors now gives a factor, and specifically
an ordered factor when combining ordered
factors with identical levels.
```

In R versions before 4.0.0, by default, `data.frame()` and `read.table()` convert all non-numerical values into factors.

This can be useful, or (more often…) it can be annoying.

Options to change this behaviour:

- `stringsAsFactors=FALSE`, or
- `as.is=TRUE` (for `read.table` only)

It can also be set by default using

```
options(stringsAsFactors=FALSE)
```

But this is not recommended, as your code may not work anymore if someone else uses it without specifying the same default option.

## CHANGES IN R 4.0.0
## SIGNIFICANT USER-VISIBLE CHANGES

```
R now uses a stringsAsFactors = FALSE default,
and hence by default no longer converts
strings to factors in calls to data.frame()
and read.table().

A large number of packages relied on the
previous behaviour and so have needed/will
need updating.
```

## *Factors and memory size*

In old versions of R, using factors for long vectors could save memory :

```
> f1 <- sample( c("Homo Sapiens", "Mus Musculus"), 10000,
                replace=TRUE)
> summary(f1)
   Length     Class      Mode
    10000 character character
> table(f1)
f1
Homo Sapiens          Mus Musculus
        4945                  5055
> object.size(f1)
80168 bytes
> f2 <- factor(f1)
> object.size(f2)
40544 bytes
```

## *Factors and memory size*

In recent versions of R (2.6+) it is not the case anymore, as R stores only once each occurrence of a string in a vector:

```
> f1 <- sample( c("Homo Sapiens", "Mus Musculus"), 10000,
                replace=TRUE)
> summary(f1)
   Length     Class      Mode 
    10000 character character 
> table(f1)
f1
Homo Sapiens              Mus Musculus
        4945                      5055
> object.size(f1)
40104 bytes
> f2 <- factor(f1)
> object.size(f2)
40312 bytes
```

## What do these commands do ?

```
c=c(c=c)
```

```
c=c(c="c")
```

## What do these commands do ?

```
C=C(C=C)
```

## Let’s read from right to left

C=C(C=C)

## Let’s read from right to left

C=C(C=C)

```
c=c(c=c)
```

```
> c
function (...)  .Primitive("c")
```

# The `c()` function, as an object

`c=c(c=c)`

```
> c
function (...)  .Primitive("c")
```

# Let’s read (more or less) from right to left

C=C(C=C)

## Let’s read (more or less) from right to left

C=C(C=C)

## Calling the `c()` function, in order to create a vector

```
c=c(c=c)
```

## Calling the `c()` function, in order to create a vector

```
c=c(c=c)
```

The vector contains a single element: the “c” object

C=C(C=C)

C=C(C=C)

## Giving the name «c» to the element of the vector

```
c=c(c=c)
```

C=C(C=C)

C=C(C=C)

## Storing the vector in a new «c» object

```
c=c(c=c)
```

## The final result

```
c=c(c=c)
```

```
> c=c(c=c)
> c
$c
function (...)  .Primitive("c")
```

## What about the other one ?

```
c=c("c")
```

```
> c=c("c")
> c
  c
"c"
```

## One more question…

## One more question…

**How does R store both the `c()` function and the `c` vector, and how does it differentiate between them ?**

```
> c=c(c=c)
> c=c(c="c")
> c
  c
"c"
```

**We will come back to this later !**

## *Reminder: getting information about R objects*

The `summary()` command gives some brief information about an R object; its output depends on the type of object:

```
> summary(mylist)
       Length Class  Mode
ages   4      -none- numeric
height 4      -none- numeric
sex    4      -none- character
```

```
> summary(aov( measure ~ groups ) )
            Df Sum Sq Mean Sq F value Pr(>F)
groups       1   0.09   0.088   0.018  0.893
Residuals   28 134.85   4.816
```

*Reminder: getting information about R objects*

## How does `summary()` know what to print for different objects ?

```
> summary(mylist)
       Length Class  Mode
ages   4      -none- numeric
height 4      -none- numeric
sex    4      -none- character
```

```
> summary(aov( measure ~ groups ) )
            Df Sum Sq Mean Sq F value Pr(>F)
groups       1   0.09   0.088   0.018  0.893
Residuals   28 134.85   4.816
```